Using Virtual Lines to Enchance Locality Exploitation *

O. Temam, Y. Jegou †

Abstract

Because the spatial locality of numerical codes is significant, the potential for performance improvements is important. However, large cache lines cannot be used in current on-chip caches because of the important pollution they breed. In this paper, we propose a hardware design, called the *Virtual Line Scheme*, that allows the utilization of large *virtual* cache lines on when fetching data from memory for better exploitation of spatial locality, while the actual *physical* cache line is smaller than currently found cache lines for better exploitation of temporal locality. Simulations show that a 17% to 64% reduction of the average memory access time can be obtained for a 20-cycle memory latency. It is also shown how simple software informations can be used to significantly decrease memory traffic, a flaw associated with the utilization of both large physical or large virtual cache lines.

Keywords: cache architecture, spatial locality, temporal locality, numerical codes.

1 Int oduction

Due to the constantly decreasing processor cycle time, the average memory access time observed by processors is rapidly increasing, because of high network or memory latency. Consequently, the cost of a cache miss can become prohibitive for monoprocessor and multiprocessors, urging for the development of hardware or software solutions.

Numerical codes can be particularly sensitive to cache performance because their working set is usually large, resulting in numerous memory accesses. Because of array references, numerical codes generally have strong spatial and temporal locality properties. While temporal locality can be exploited with loop blocking [2, 12, 16], little support (except for the cache line) or flexibility is available for the exploitation of spatial locality. However, experiments [15], tend to show that the optimal cache line is not the same for all codes, depending on whether spatial or temporal locality should be preferably exploited.

Small direct-mapped caches are now popular [13, 4] because the on-chip space used and the cache hit time can be minimized (the cache hit time should be minimized if it determines the processor cycle time). However, such caches can be very sensitive to cache conflicts and pollution phenomena, preventing the use of large cache lines.

In this paper, we propose a cache design, called the Virtual Line Scheme (VLS) that allows the utilization of large cache lines in small direct-mapped caches, while avoiding most of the flaws associated with large cache lines: increased cache conflicts and cache pollution. The scheme proposed gives the illusion that a variable line size can be used for each reference within a code. The actual physical cache line size is smaller than usual cache line sizes in order to better exploit temporal locality, while large virtual cache lines are used in order to better exploit spatial locality. Simulations show that the mean average memory access time observed over all benchmarks is equal to 1.7 cycles per reference for a 20-cycle memory latency. It is also shown how simple software informations can be used to significantly decrease memory traffic, a flaw common to both large physical and large virtual lines.

In section 2, the tradeoff between spatial locality and temporal locality exploitation is discussed. In section 3, the principles of the virtual line scheme is presented and performance measurements are provided. In section 4, it is shown how to assist the virtual line scheme with software informations for decreasing the memory traffic. Finally, in section 5, design and implementation issues are discussed.

2 Exploitation of spatial and tempo al locality

The tradeoff between spatial and temporal locality The principles of caches is to exploit the temporal and spatial locality that reside in codes. Temporal locality is exploited by keeping data in cache, while spatial locality is exploited by loading several consecutive data, i.e. a cache line, on each memory request. The choice of the optimal line size is a delicate tradeoff between temporal and spatial locality exploitation.

With respect to temporal locality, the number of cache entries should be as large as possible so as to minimize *cache pollution* (data loaded in cache that is not used) and *cache conflicts* (data competing for the same cache locations). Since the number of cache entries is equal to the ratio $\frac{cache\ size}{cache\ line\ size}$, the cache line size should be chosen as small as possible.

*This work was funded by the BRA Esprit III Projet APPART, European agency DGXIII.

†*High Performance Computing Division*, Department of Computer Science, University of Leiden, The Netherlands.

With respect to spatial locality, the number of elements brought in cache in one time, i.e. the cache line size, should be as large as possible, in order to minimize the number of memory requests, thereby reducing the average memory access time to an element.

```
DO J1=1,N1
  reg = A(J1)
  DO J2=1,N2
    reg += B(J2)*C(J2,J1) + D(J1,J2)
  ENDDO
  A(J1) = reg
ENDDO
```

Figure 1: *Example of numerical loop nest*

Reducing cache pollution vs. exploiting spatial locality Let us illustrate this tradeoff with the example of figure 2. References to A, B, D all exhibit flawless spatial locality (column-wise storage is assumed for C and D). So, apparently, for these references, the larger the line size the smaller the number of cache misses. Note that array B exhibits temporal locality, therefore it is necessary to minimize cache pollution so that array B can be kept in cache. However, reference to C does not exhibit spatial locality (actually, it does, but the time between reuse of a cache line being equal to one execution of loop J_2, it is unlikely this spatial locality can be exploited). Consequently, if the cache line size is equal to l_S words, then l_S words of array C are loaded in cache on each iteration of J_2, while l_S words are loaded every l_S iterations for the other references (due to their spatial locality). So, *array C is going to pollute the cache l_S times faster than array D*. Besides, the reference to C misses on each iteration of J_2, while the reference to D misses one every l_S iterations.

Ideally, a very small line size should be used for reference C, while other references should use a large line size. This would not reduce the number of misses to array C, but it would considerably decrease the cache pollution due to this array, and thereby increase the hit ratio of array B. Note that array D also pollutes the cache by bringing data without temporal locality. However, the gains obtained by exploiting the spatial locality of array D will generally outperform the cache pollution it induces.

Cache line size as a performance bottleneck In numerical codes, the ability to use large cache lines could significantly increase performance. Indeed, array references are often stride-1 accesses [17], and besides, arrays without temporal locality are also commonly found (like a matrix access in a matrix-vector multiply primitive). The only way to reduce the number of cache misses of such references (called *cold-start misses*) is to use large line sizes. This is all the more true that efficient algorithms now exist for exploiting temporal locality [2, 12, 16], so that such misses can become the performance bottleneck of numerical loop nests. Assume for instance, that in the loop nest of figure 2, the reference to C is $C(J_1, J_2)$ (stride-1 access). Then, if N_2 is small enough so that B fits in cache, the number of cache misses is approximately equal to $\frac{2N_1N_2}{l_S} + N_1 + \frac{N_2}{l_S}$ where the first term corresponds to C and D, the second term to A (assuming the spatial locality of A cannot be exploited due to the reuse distance), and the third term to B. Then, doubling the line size nearly decreases by two the number of misses of this loop nest.

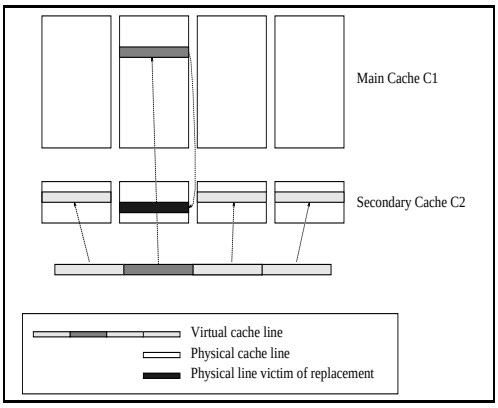


Figure 2: *Virtual line scheme*

Non-numerical codes On the contrary, non-numerical codes (like Unix tools) typically exhibit an irregular pattern of accesses to memory. Therefore, though they can exhibit spatial locality, they mostly favour small line sizes for a better exploitation of their temporal locality (i.e. for minimizing cache conflicts and cache pollution).

3 The vi tual line scheme

The *Virtual Line Scheme* is a solution for eliminating the tradeoff between small cache lines for temporal locality, and large cache lines for spatial locality. On a cache miss, large chunks of elements are fetched from memory (i.e. a *virtual cache line*), but only a few are actually stored in cache (i.e. a *physical cache line*). Therefore, spatial locality is exploited but does not induce cache pollution.

3.1 Principles

In this scheme, the *physical line size*, i.e. the actual cache line size l_S is either or the same order or smaller than current cache line sizes (between 8 and 32 bytes). On a memory request, a large *virtual* line of size $L_S = n \times l_S$, i.e. made of n physical lines, is fetched from memory, but only the physical line containing the requested word is actually loaded into cache. The remaining physical lines of the virtual line are loaded into a small secondary cache. The same way a virtual line is divided into n sub-lines, the primary and secondary caches are divided into n banks (see figure 2).

The *Virtual Line Scheme* should not be confused with *subblock placement* (see [6]). In *subblock placement*, there is only one tag for all the subblocks, while in the *Virtual Line Scheme*, there is one tag for each physical line (i.e., physical lines are not subblocks, they are effective cache lines). The purpose of *subblock placement* is to reduce the miss penalty associated with long cache lines and the on-chip space used by address tags. The purpose of the *Virtual Line Scheme* is to allow more flexible use of the cache space for better temporal locality exploitation by reducing the cache line size, and by *virtually* (not physically as for *subblock placement*) grouping together cache lines in order to exploit spatial locality (such a group then corresponds to a *virtual line*) .

The main flaw of the *Virtual Line Scheme* is the additional on-chip space required to implement the multi-bank design and the secondary cache. In case the physical line is equal to 16 bytes for example, instead of the now usual 32 bytes, then double the number of tags is necessary. This can increase the cache size by approximately 10% for a physical line of 16 bytes compared to a physical line of 32 bytes (assuming a 20-bit tag per cache line). The secondary cache also induces an on-chip space overhead, however two facts make it a good tradeoff. First, the secondary cache can be used as a write buffer as explained in section 5. Second, it can also be used as a victim cache as discussed below and in section 5. Though victim caches have not yet been implemented, there are clear evidence [11] of the performance improvements they can bring.

The secondary cache and the additional tags are the main overhead associated with the implementation of the virtual line scheme. Other less significant hardware add-ons must be included that are detailed in section 5.

3.1.1 orkings of the virtual line scheme

Main parameters In the remainder of the paper, the main cache is called C_1 and the secondary cache is called C_2. C_1 has the characteristics of several current on-chip caches, i.e. cache size is equal to 8192 bytes and it is direct-mapped [13, 4], while the secondary cache is 1024 bytes large and is 4-way associative. The parameters of the secondary cache have been determined experimentally, and other sizes and associativity can be used as shown in section 5.

Because C_1 is direct-mapped and C_2 is set-associative, the hit time of C_1 is considered equal to one cycle, while the hit time of C_2 is equal to two cycles.

A physical line size of 16 bytes has been chosen, and the virtual line size is taken equal to 64 bytes (see section 5), i.e. both caches are divided into 4 banks.

The virtual line scheme algorithm On a processor request, both caches are tested in the same time, only C_2 answers one cycle later than C_1:

- If the processor hits in C_1, the execution resumes and the reply of C_2 is ignored.
- If the processor misses in C_1 and hits in C_2, the processor resumes execution after two cycles. Meanwhile, the *physical line* where the hit occurred is transferred from C_2 to C_1. Besides, due to the *victim cache* role of C_2, the victim physical line of C_1 is transferred into C_2 in place of the hit line (see figure 2). For that reason, both caches are considered to be stall (with respect to the processor) again two cycles after the hit in C_2 occurred.
- If the processor misses in both caches, then a memory request is issued for a *virtual line*. As mentioned above, only the target physical line is transferred into C_1, the other physical lines of the virtual line being stored in C_2. This is nearly the only step of the mechanism which is aware of the virtual line. The rest of the time, both caches work very much alike a two-level cache hierarchy (except for the multi-bank mechanism) with a line size equal to l_S. Note that the cache miss latency is increased by one cycle over a standard cache because of the delayed answer of cache C_2.

3.1.2 Large physical line vs. large virtual line

Observed memory latency Due to the 2-cycle access time to C_2, the benefit of using a large virtual line versus a large physical line is not obvious. Accessing the 4 physical lines of a virtual line (in case $L_S = 64$ and $l_S = 16$) costs t_l cycles (where t_l is the memory latency) plus 1 cycle for the first physical line and 3×2 cycles for the remaining 3 lines in case they are all used, so a total of $t_l + 7$ cycles. For a physical line of 64 bytes, the total penalty is only equal to $t_l + 4 \times 1$ cycles. Nevertheless, all physical lines of a virtual line are not necessarily used, so that the actual average total latency for accessing large physical or virtual lines are more alike.

Besides, when the requested physical line comes back from memory, the processor can resume execution at $t = t_l + 1$ and *the primary cache is not stall*, since the remaining lines are stored in cache C_2. A standard cache would also resume execution at $t = t_l + 1$, but the primary cache would be stall for several more cycles while the new cache line is being loaded. In a superscalar processor where one request per cycle can be issued, this can result in processor stall cycles.

Solutions for avoiding processor stall during large cache line reloads have already been proposed for the IBM RS6000 [7], where a buffer is used to store the incoming line, so that the processor needs not wait for the cache reload to be performed. In the virtual line scheme, C_2 acts as such a buffer.

Cache pollution However, the most important asset of large virtual lines over large physical lines is to

minimize primary cache pollution. Indeed, a physical line of a virtual line is loaded into C_1 only if it is requested by the processor. That is why the hit ratio of the primary cache with a 16-byte cache line is nearly always equal to that of a standard cache with a 32-byte cache line (see figure 7; see section 3.2.1 for details on benchmarks). Note that for `LL`, a standard cache still performs better, because this code has very strong spatial locality; on the other hand, for `MM`, the reduction of pollution increases significantly the hit ratio of C_1. Besides, a physical line is loaded into the primary cache only *when* it is requested. This latter fact can be important, because though a reference can exhibit spatial locality, the time between reuse can be long, so that storing in cache data that is not used at once, is also a form of cache pollution.

Scarce spatial locality In case the stride of a reference is not equal to one, then the virtual line scheme can perform better than large physical lines, since among the physical lines of a large virtual line, only those used are loaded into cache C_1. But the virtual line scheme proves to be particularly useful when references exhibit scarce spatial locality such as $C(J_2, J_1)$ in example of figure 2. While a 64-byte physical line would load into cache 8 64-bit words in cache(7 corresponding to cache pollution), a 64-byte virtual line would only load 2 words (16 bytes) in cache, the other words staying in cache C_2.

Performance comparison Figure 3 shows a performance comparison between large physical lines and large virtual lines. Increasing physical line size nearly always results in performance reductions. On the other hand, the virtual line size scheme not only tolerates well large virtual lines, but for most numerical codes, the performance is increased. For non-numerical codes, the performance either slowly increases or is stable when the virtual line size increases, while it degrades rapidly when physical line size increases.

3.2 Performance of the virtual line scheme

3.2.1 Overview of experiments

Simulations have been performed cycle by cycle in order to catch all phenomena associated with the design (bus utilization, caches stall because of line transfers, slower response time of cache $C_2 \ldots$).

A collection of 10 benchmarks has been used. Though it is difficult to choose a *representative set* of benchmarks, we tried to combine different types of codes to illustrate the maximum number of behaviors. Some numerical codes have been picked in the Perfect Club Suite [1] (benchmarks `AP`,`ARC`,`BDNA`,`WS`), some codes are Unix tools (`CC` is the gnu-cc compiler, `CPR` is the compress utility, `TEX` is the LaTeX compiler), and other codes are numerical primitives (`LL` is a Lawrence Livermore loop, `SPMV` is sparse matrix-vector multiply loop, and `MM` is matrix-matrix multiply loop).

One-million references traces have been extracted for each code, while the first million references has been ignored in order to avoid non-representative references corresponding to initialization sections of codes.

3.2.2 Reduction the average memory access time

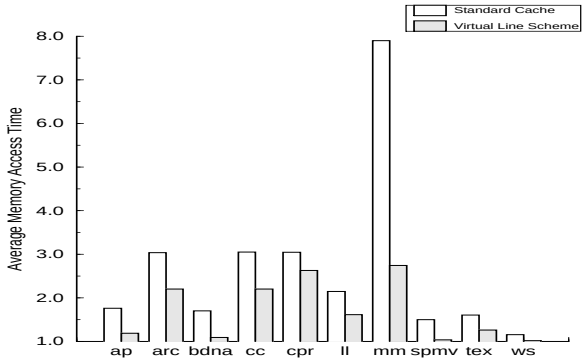


Figure 4: *Performance comparison of the* Virtual Line Scheme.

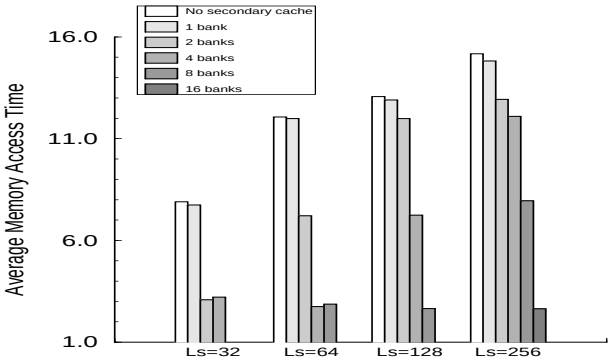


Figure 5: *Influence of physical and virtual line sizes for MM.*

For both numerical and non-numerical codes, significant performance improvement is observed over a standard 8192-byte cache with a line size of 32 bytes (see figure 4).

The reduction of the average memory access time varies between 17% and 64%. The mean average access time is equal to 1.7 cycles against 2.7 cycles for a standard cache. The performance improvement is most spectacular for MM (Matrix-Matrix Multiply), because one of the two main references of the innermost loop nest is non stride-1, while the other reference exhibits flawless spatial locality. Consequently, in a standard cache the pollution is all the more important that the cache line is large, while with the virtual line scheme, large virtual lines induce no additional pollution as far as the physical line is kept small (see figure 5). The matrix-matrix multiply example exploits both properties of the virtual line scheme: increased exploitation of spatial locality, and reduced cache pollution.

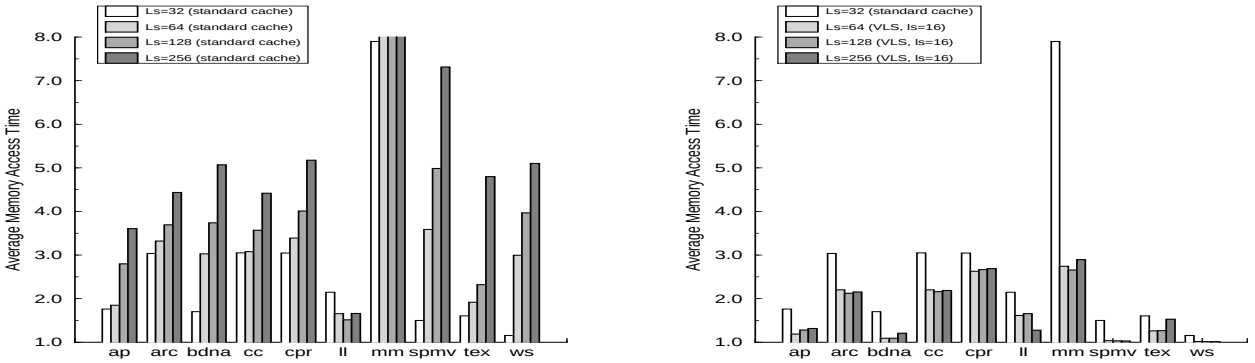
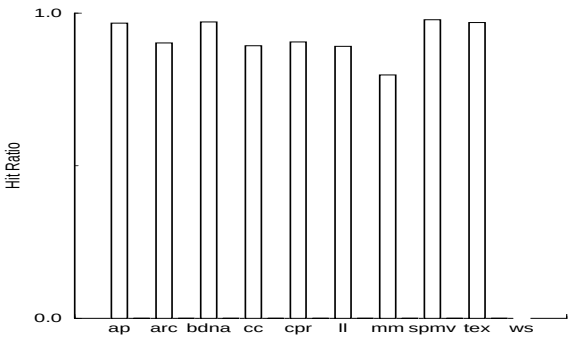


Figure 3: *Performance of large virtual and physical lines.*

Otherwise, it can be noticed that the average memory access time is smaller with the virtual line scheme than with a standard cache for all codes (see figure 4), which means the design is *safe*, i.e. it does not degrade the performance of some codes (at least for the benchmarks used, and the memory latency value considered).

3.2.3 Increased performance by reduction of cache pollution and exploitation of spatial locality

The virtual line scheme can be profitable either by limiting cache pollution or by allowing exploitation of spatial locality.



Temporal locality exploitation In figure 8, the fraction of words prefetched (i.e. fetched from memory into C_2, because they did not belong to the requested physical line) that were effectively used is indicated. As can be seen, a numerical code like `ARC` that benefits from the virtual line scheme uses a negligible number of prefetched data. On the other hand, figure 7 shows that the hit ratio of C_2 is not significant. This paradoxical behavior indicates that the reduced cache pollution due to the smaller physical line and the secondary cache allow a better exploitation of temporal locality. Note that a small physical cache line size also increases the efficiency of the secondary cache. This type of phenomenon was more expected for non-numerical codes (see benchmark `CPR`) rather than for numerical codes.

Spatial locality exploitation More naturally, the virtual line scheme also performs well by simply exploiting spatial locality. In figure 8, it can be seen that benchmarks `BDNA`,`LL`,`MM`,`SPMV` or `WS` use more efficiently the words prefetched. The efficiency increases when the physical line size decreases, but recall that double the number of accesses to C_2 are necessary when the physical line size is divided by two. Therefore, the number of hits to C_2 artificially doubles while the number of hits to C_1 remains constant. Still, when the secondary cache is intensively used as both a prefetch buffer and/or a victim cache, the hit ratio in C_2 can be

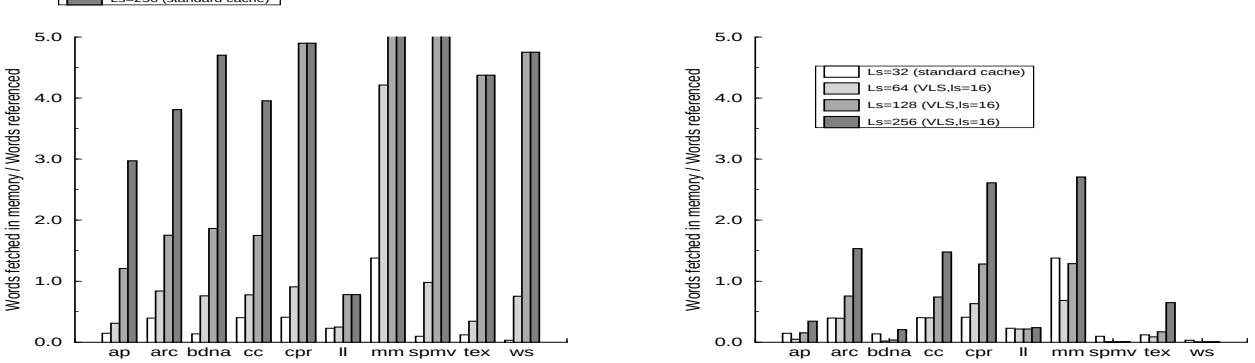


Figure 6: *Memory traffic for large virtual and physical lines.*

were used, the size of the line would have no impact on the memory traffic, but figure 8 shows that, though this is true for some codes (`LL`,`SPMV`,`WS`), it is not true for other codes (`ARC`,`CC`,`CPR`,`MM`).

Selective miss requests However, the virtual line scheme has been designed so as to minimize memory traffic. When a cache miss is issued, *only the physical lines of the virtual line that are not present in C_1 are fetched from memory.* Indeed, due to the loose connection between the physical lines of a virtual line, it is possible that a miss occurred on a physical line while other physical lines of the same virtual line are still present in cache. Fetching these lines from memory would not only induce coherence issues, but it would be wasteful. Therefore, the presence of the complementary physical lines in C_1 is tested before issuing the miss request (note that the same problem arises for cache C_2; this issue is discussed in section 5).

Implementation issues However, it can be argued this facility may increase the latency. Two solutions can be adopted to avoid that. First, all physical lines can be tested simultaneously, since they necessarily belong to different banks and have a tag each. On the other hand, some additional hardware is necessary to perform that operation. Otherwise, another cheaper solution can be applied: when a miss request is issued, the time necessary to spend the request for the first word is significantly larger than for subsequent words. Therefore, it is conceivable that when this first word request is being processed, the cache is tested for the presence of the other physical lines. This solution has the advantage of limiting the amount of the hardware required.

Reduction of memory traffic Using any of the two methods (the choice is transparent to the simulations), it can be observed that the memory traffic increases significantly less with the virtual line scheme than with large physical lines (see figure 8). Overall, the increase of the memory traffic remains reasonable (for $L_S = 64$), but some non-numerical codes like `CPR` still exhibit a memory traffic increase of approximately 30%, though their average memory access time decreases. For larger virtual lines, though the average memory access time remains competitive, the memory traffic is more significantly increased.

3.3 Prefetching

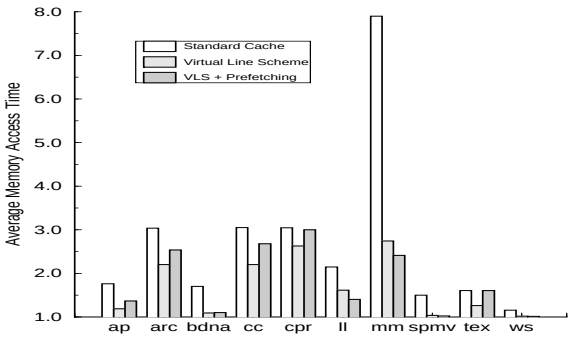


Figure 9: *Performance of VLS+Prefetching.*

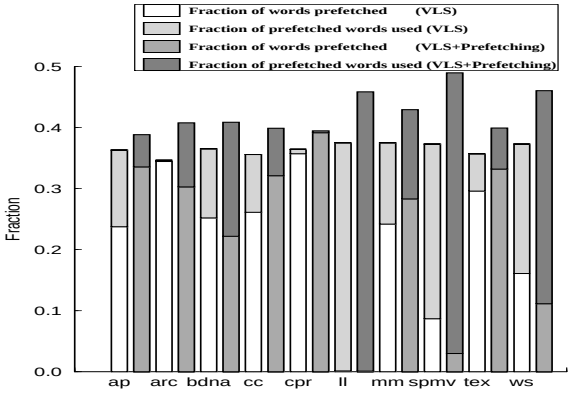


Figure 10: *Memory traffic for VLS+Prefetching.*

Though the virtual line scheme allows the utilization of large line sizes, excessively large sizes could induce severe bus contention and secondary cache pollution, reducing its efficiency both as a buffer and as a victim cache (see figures 3 and 6). Besides, the larger the line size, the smaller the efficiency, i.e. the smaller the ratio of words fetched over words used as seen

above. Finally, large line sizes do not prevent periodic cache misses during a vector access.

Stride prefetching An alternative to large line sizes has been proposed in [9, 3, 14, 10]. The principle is to detect the accesss stride of a reference, then predict the next address to be used based on the address being currently referenced and the stride, and then prefetch the corresponding data. Though simulations proved the efficiency of such schemes, they require relatively heavy and complex implementations. Besides, array references within a loop nest are often stride-1 (see [17]), making complex stride detection mechanism less necessary.

Combining the *virtual line scheme* with prefetching The virtual line scheme actually constitutes a convenient architecture base for introducing simple stride prefetching. Consider a physical line of a virtual line that has been stored in one of the banks of C_2 because it was not requested by the processor, and which first word corresponds to virtual address A. If this line is used later on, then the *physical line* corresponding to word $A + L_S$ is prefetched, i.e. the next physical line corresponding to the same bank. Each bank then behaves independently from other banks. Consequently, even stride accesses can be exploited, up to strides equal to the size of a virtual line. Note that only the necessary physical line is loaded instead of a whole virtual line.

With respect to implementation, only one bit is necessary per cache line of the secondary cache to indicate whether this line was prefetched or victim of replacement. Otherwise, a buffer is necessary to store the data address of the next physical line to be prefetched, in case memory is busy when a prefetching request needs to be issued.

Performance and efficiency So, for vector accesses, a miss occurs only for the first virtual line, then the next physical lines are prefetched. Note that if the regularity of accesses is disrupted, only one physical line in excess is loaded into secondary cache. Indeed, if the physical line that was fetched in C_2 is not used, the next line will never be prefetched, since prefetch occurs only when a line of C_2 (that was never used before) is transferred in C_1. This principle limits the amount of additional memory traffic and wrong predictions. Besides, as for the mechanisms proposed in [9, 3, 14, 10], regularity appearing in complex codes (non-rectangular loops, loops with *if* statements) can also be exploited.

As can be seen on figure 9, prefetching can increase the performance of codes with strong spatial locality (`LL`,`MM`) but degrades the performance of other codes (with respect to the standard virtual line scheme, not with respect to a standard cache). These performance degradations are first due to *the additional memory traffic* which increases the observed memory latency (in case a cache miss requests is pending because a prefetch request locks the memory), and also because of the additional data fetched in the secondary cache *which degrade its efficiency as a victim cache* and sometimes even as a prefetch buffer (see figure 10, benchmark `AP`).

Note however, that the fraction of prefetched words that are used nearly often increases (see figure 10, benchmarks `ARC`, `BDNA`...), which means prefetching has the potential to improve performance, as far as its side-effects (increased memory traffic and secondary cache pollution) can be eliminated.

Related work The virtual line scheme combined with prefetching is also close to the *multi-way stream buffers*

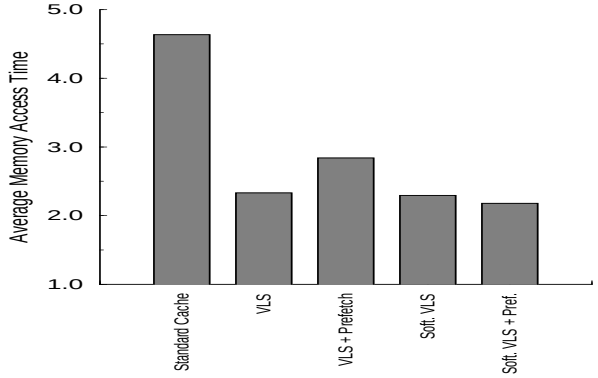


Figure 11: *Performance of software-directed VLS (for example of figure 1).*

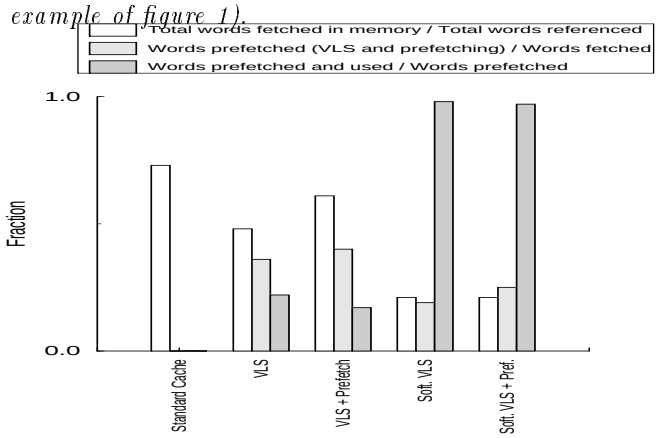


Figure 12: *Memory traffic for software-directed VLS (for example of figure 1).*

erence exhibits spatial locality. This information could be used by the virtual line scheme to decide whether a large virtual line or a small physical line should be loaded for the corresponding reference. This is all the more possible that data locality optimizing algorithms [2, 12, 16] are now sufficiently mature and powerful to detect temporal and spatial locality in numerical codes. Besides, as suggested in [5], very simple informations are sufficient to coordinate a combined software/hardware mechanism for exploiting locality. In our case, only one bit per load/store instruction is sufficient to specify whether the reference exhibits spatial locality.

Example Consider again the example of figure 2. Reference $C(J_2, J_1)$ has spatial locality while reference $D(J_1, J_2)$ has no spatial locality. Therefore, for reference $D(J_1, J_2)$, a virtual line of four physical lines is loaded, while for reference $C(J_2, J_1)$, only a single physical line is loaded. The first consequence is a reduction of the secondary cache pollution inducing a reduction of the average memory access time, as shown on figure 11. The second and more important effect, is a significant reduction of the memory traffic as shown on figure 12. Note that the experiments were restricted to an example, because of the unavailability of a compiler for instrumenting a code with spatial locality information.

The underlying idea of assisting the virtual line scheme in deciding whether a physical or a virtual line should be fetched is to effectively implement a *variable cache line size* mechanism (and not just give the illusion to cache C_1 that such a mechanism is used). It is possible to further refine this system, by specifying the number of physical lines that should be brought instead of choosing between one physical line and one virtual line. The general purpose is to provide the necessary hardware support for developing software optimization of spatial locality, because the hardware design of current caches provides no flexibility for performing such optimizations.

Assisting temporal locality exploitation Further on, the structure of the virtual line scheme can also be convenient to assist the exploitation of temporal locality as well. Cache pollution and conflicts can be further reduced, if data deprived of temporal locality is not allowed to reside in the primary cache, or at least not after it has been used. The underlying idea is to impose priorities between data based on their temporal locality properties. This principle has already been investigated in [5], but the performance of the scheme was limited by the absence of an alternative cache where to store non-reusable data, in order to avoid cache pollution, and the fact temporal locality was favoured over spatial locality. Such a technique can also be compared with bypassing, which has even been implemented in some processors like the i860 [8]. However, bypassing prevents exploitation of spatial locality, unless a dedicated buffer is used to store bypassed cache lines. The secondary cache of the virtual line scheme could be used for these purposes as far as information is available on the temporal locality of each reference.

This information can then be used to assist the victim cache process. If the reference has no temporal locality, then the line is not transferred to the secondary cache when it is victim of replacement. Note that this information is not used to bypass the cache. As mentioned above, this is not efficient since the spatial locality of a reference cannot be exploited. Consider example of figure 2. Reference $D(J_1, J_2)$ has no temporal locality but strong spatial locality. Therefore, the reduction of pollution obtained by bypassing the cache would probably not compensate for the added memory accesses. Therefore, when a cache line of reference $D(J_1, J_2)$ is transferred into C_1, the victim line is transferred into C_2 (if it has temporal locality). When the line of $D(J_1, J_2)$ is victim of replacement in turn, it is discarded instead of being transferred into C_2. As a result, the pollution of cache C_2 by $D(J_1, J_2)$ remains minimum.

Prefetching

Note that software control can also be used for prefetching. It has been seen in section 3, that sometimes prefetching can bring performance improvements, but that it often severely increases memory traffic which makes it an unsafe technique. A solution is to start prefetching for a reference only if spatial locality has been detected (an information provided by the spatial locality bit). In that case, it is unlikely many wrong predictions would occur. In figure 11, it can be seen that a combination of software-directed virtual line scheme and prefetching performs significantly better than standard virtual line scheme with prefetching. The best performance is actually obtained in this case. As can be seen on figure 12, this performance improvement is due to a better efficiency of prefetching and consequently a reduction of memory traffic.

Implementation issues

The informations to be provided, i.e. the presence of spatial and temporal locality for each reference of a loop nest, can be extracted at compile-time. Only two bits are necessary to store this information. The instruction set can be extended to accommodate such new load/store informations.

When the compiler is unable to provide informations on the locality of the reference, an alternative is to set by default that the data has temporal locality. Experiments proved this solution to be more efficient than no temporal locality by default. Regarding spatial locality, it is also possible to set the spatial locality bit on by default, since loading only one physical line per memory request proved to be inefficient. When some codes are known to exhibit little spatial locality, it would be possible to set the default virtual line size to smaller values like 32 bytes.

5 Design and implementation issues

5.1 Physical line and virtual line sizes

The size of the elementary physical line and of the virtual line are tightly coupled. The tradeoff is to choose a physical line that is small enough to maximize the number of cache entries (i.e. cache size over line size) in order to better exploit temporal locality, but also to find a physical line that is large enough so that the number of accesses to the secondary cache are not too frequent (the hit time in C_2 being longer).

Similarly the tradeoff for the virtual line is to find a line size that is large enough so that sufficient spatial locality can be exploited, but that is small enough so that an important fraction of the words fetched are used, thereby minimizing excessive memory traffic.

Experiments proved that relatively large virtual line sizes can be considered (see figure 3), which is promising because it means spatial locality can be efficiently exploited. Still, a virtual line size of 64 bytes has been selected because it induces only moderate memory traffic increase.

On the other hand, the choice for the physical line size is more restrictive. The optimal value seems to be equal to 16 bytes, though in some cases, physical line sizes of 8 bytes are preferred. However, mostly codes without spatial locality benefit from such small lines, the performance of codes with spatial locality being degraded by numerous accesses to the secondary cache (note that 8 bytes corresponds to one double precision floating point number).

5.2 Secondary cache

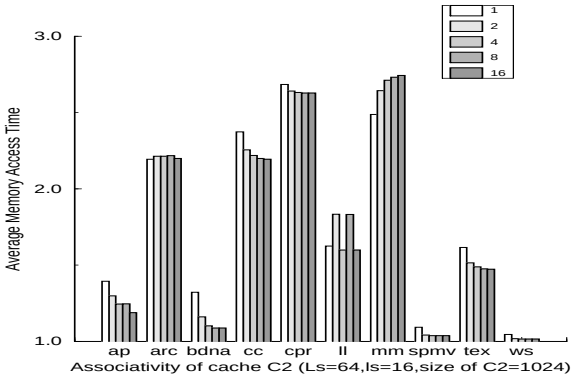


Figure 13: *Variation of VLS performance with the associativity of cache C_2.*

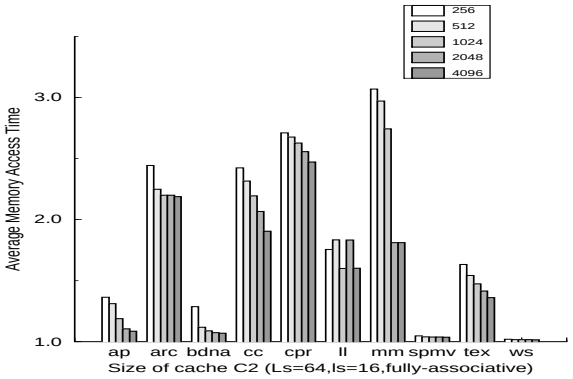


Figure 14: *Variation of VLS performance with cache C_2 size.*

Size and associativity

The characteristics of the secondary cache to be determined are the size and the associativity. Finding an optimal tradeoff is difficult, because the performance of the whole mechanism is either stable or varies nearly linearly when the secondary cache size or associativity increases. Since the secondary cache size is small anyway, it is not surprising that cache conflicts are numerous, and therefore that performance improves constantly when associativity or size is increased. Because of implementation constraints more than for performance reasons, the secondary cache size has been limited to 1024 bytes and has been chosen 4-way associative.

rite buffer mechanism

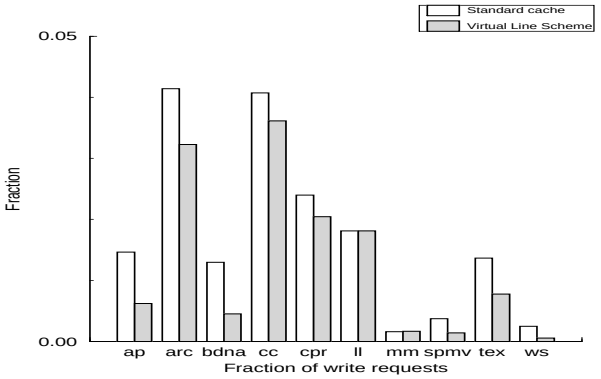


Figure 15: *Amount of write back requests.*

As mentioned in section 3, the secondary cache acts as a write buffer, sparing some on-chip space. The mechanism relies on the replacement policy of the secondary cache. An LRU policy is used. In each bank of the secondary cache, when a physical line that is dirty reaches the lowest priority with respect to LRU, it is written back to memory. Therefore, only the data that have not been used for a long time, i.e. that have a fair chance not to be reused, are written back to memory. This mechanism can significantly reduce the number of memory write requests (see figure 15), which could decrease the burden on coherence protocols in a multiprocessor.

5.3 Coherence issues

Coherence in the secondary cache As mentioned in section 3, coherence issues can arise because of the two-level cache hierarchy. It has been shown that coherence can be maintained in the primary cache by testing for the presence of the physical lines of a virtual lines before fetching these lines from memory. For the secondary cache, the same problem arises and two solutions can be adopted. First, the same technique used for C_1 can be employed for C_2, but the miss latency must be increased by one cycle, the access time of the secondary cache being longer. Another solution is to send the memory request without testing the secondary cache beforehand. Cache C_2 is tested while the request is being processed. Then, for each bank, a flag bit is set if the physical line is found. When the data returns from memory, the physical lines corresponding to a bank where the bit is set are discarded. Note that it would be more simple to directly invalidate the data in the secondary cache, but since it can be dirty this would not be always possible.

Prefetching

The most difficult problem is to maintain coherence when prefetching is used. When a physical line is prefetched and stored in C_2, it is not possible to test for its presence in C_1 in order to check coherence. Such a test would stall C_1, which could mean stalling the processor, especially if the processor is superscalar (the cache then being often solicited). Since prefetching requests can be numerous, this is not tolerable.

The solution adopted is to store the physical line in C_2 without checking C_1. Since the secondary cache acts as a write buffer, further coherence issues can then be eliminated. Indeed, a dirty physical line in C_1 needs to go through cache C_2 (the write buffer) before being written back to memory. Before transferring a dirty line from C_1 to C_2, cache C_2 is tested for the presence of the same physical line. If that occurs, this line is immediately invalidated, and the line of C_1 is then stored in C_2. Note that it is impossible that the line in C_2 is dirty, since writes can only occur in C_1. Because this mechanism can increase swap time during a victim cache operation, it is possible to further refine it. When a processor request hits in C_1 and hits in C_2, then redundancy is detected and the data in C_2 is invalidated (note that it cannot be dirty).

5.4 Comparisons with large and associative caches

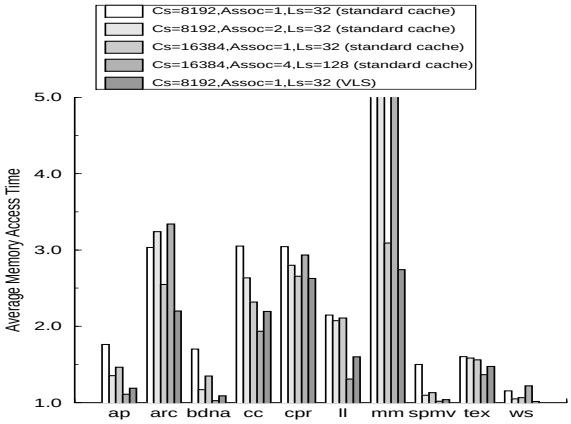


Figure 16: *Comparison of VLS with different cache architectures.*

Due to the on-chip space required to implement the virtual line scheme, it is necessary to compare the scheme performance with larger caches. It can be seen on figure 16, that a 16-kbyte direct-mapped cache has either comparable (for non-numerical codes) or smaller (for numerical codes) performances than the virtual line scheme, which indicates that available on-chip space should be preferably dedicated to such hardware optimizations.

The performance of set-associative caches has also been investigated because they provide an indication of the amount of cache conflicts that can be removed. As seen on figure 16, only a 16-kbyte 4-way associative cache with a 128-byte cache line (as in the IBM RS6000 [7]) can outperform the virtual line scheme. Though a 2-way 8-kbyte cache also increases significantly the performance of non-numerical codes, it is less efficient at

exploiting spatial locality and therefore performs worse for numerical codes.

Note that for the experiments, the hit time of the set-associative caches has been taken equal to one cycle, which is optimistic.

6 Conclusions

In this paper, a hardware scheme for using large cache lines without the usual associated flaws has been proposed. Simulations showed the efficiency of the scheme for both numerical codes and non-numerical codes, because large virtual lines allow a better exploitation of spatial locality, while small physical lines and the secondary cache allow a better exploitation of temporal locality. Inspite of the increased memory traffic, no code exhibited performance degradations. A simple software solution for reducing the memory traffic has been investigated. Though, it proved to be efficient, extensive testing is still necessary to fully validate the concept.

The underlying idea of the *Virtual Line Scheme* was to propose a cache design that provides more flexibility for exploiting both spatial and temporal locality, than current cache architectures. This design is intended as an architecture base on which further hardware or software optimizations can be developed, because it provides a convenient environment for many solutions being currently investigated (cache bypassing, data priorities, prefetching...). Our next goal is to show how the performance of current data locality optimizing techniques can be significantly enhanced if proper hardware support is provided for minimizing cache conflicts and exploiting spatial locality.

Refe ences

[1] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputing Performance Evaluation and the Perfect Benchmarks. In *Supercomputing '9* , pages 254–266, 1990.

[2] Christine Eisenbeis, William Jalby, Daniel Windheiser, and Francois Bodin. A Strategy for Array Management in Local Memory. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing*, 1990.

[3] John W. C. Fu, Janak H. Patel, and B. L. Janssens. Stride Directed Prefetching in Scalar Processors. In *MICRO-26*, 1992.

[4] J. Heinrich G. Kane. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[5] Elana D. Granston and Alexander V. Veidenbaum. An Integrated Hardware/Software Solution for Effective Management of Local Storage in High-Performance Systems. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 83–90, August 1991.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[7] IBM Corporation. *IBM RISC system/6 technology*, 1990.

[8] Intel Corporation. *Intel i86 reference manual*, 1991.

[9] T-F. Chen J-L. Baer. An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty. In *Proceedings of IEEE Supercomputing*, 1991.

[10] Y. Jegou and O. Temam. Speculative Prefetching. In *Proceedings of the ACM International Conference on Supercomputing*, 1993.

[11] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small, Fully-Associative Cache and Prefetch Buffers. In *International Symposium on Computer Architecture*, pages 364–373, May 1990.

[12] Ken Kennedy and Katherine S. McKinley. Optimizing for Parallelism and Data Locality. In *Proceedings of the ACM International Conference on Supercomputing*, pages 323–334, July 1992.

[13] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[14] I. Sklenar. Prefetch Unit for Vector Operations on Scalar Computers. *Computer Architecture News*, September 1992.

[15] Alan Smith. Cache Memories. *Computing Surveys*, 14(3), September 1982.

[16] Michael Wolf and Monica Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26(6), pages 30–44, June 1991.

[17] She Zhiyu, Zhiyuaun Li, and Pen-Chung Yew. An Empirical Study on Array Subscripts and Data Dependendencies. Technical Report 840, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, August 1989.